Zcash & Zerocash: Data mining, cryptanalysis and key reuse

Killian Davitt, Lena Petersen

March 25, 2018

Intro

This document was written as part of the GA18 Cryptanalysis course to explain my work as part of Project PK. We will discuss Zerocash, Zcash, The Zcash client, How we mined interpreted Zcash data and finally how we searched for duplicated random values in ECDSA signatures and reused public keys.

Zerocash

Zerocash is a cryptocurrency template. While mostly being similar to bitcoin, it both attempts to provide facility for highly anonymous transactions in cryptocurrency. We will not discuss Zerocash much here, instead we will focus on Zcash, it’s most prominent implementation. The prominent goal of both however is to facilitate highly anonymous cryptocurrency transactions using zero knowledge proofs.

Zcash

Zcash is one particular instance of the Zerocash protocol. It is mostly similar to the Zerocash protocol as described in but has some slight but significant differences. Zcash currency is denoted as ZEC.

Before discussing the format of anonymous transactions, it is worth noting that Zcash also includes the “Equihash” hash function as part of it’s proof of work which is designed to use a memory hard problem. The equihash function requires a certain amount of RAM to compute and thus it is expected to be far more difficult to run the equihash function on large scale ASICs which is why it was selected for Zcash.

One distinct problem with Zerocash is the time it takes to generate the zero knowledge proofs needed for highly anonymous transactions. It can take up to 1 minute to generate an anonymous transaction. Thus Zcash in it’s implementation of Zerocash allows for both highly anonymous transactions as well as standard transactions similar to bitcoin which do not use zero knowledge proofs. If a user does not have the time to wait to create a zero knowledge proof they can simply opt to create a standard non-anonymous transaction. The two types of transactions are known as “Shielded” and “Un-Shielded” and are interoperable; i.e. A user can receive funds through an unshielded transaction and then immediately send those same funds in a shielded transaction to someone else.
Structure of transactions

Firstly we will focus on unshielded transactions as shielded transactions are quite complex and will require a great deal of effort to explain.

Unshielded Transactions

The following image shows an example of an unshielded transaction in JSON. The structure is quite similar to a bitcoin transaction.

The inputs being consumed are described in “Vin” and the outputs of the transaction are described in “Vout”. The empty field “VJoinSplit” is populated in shielded transactions only.

Aside from \( n \) Vin blocks and \( m \) Vout blocks, a unshielded transaction also contains the transaction id, version and locktime.

1. **Vin**: Quite similarly to bitcoin, vin’s describe the source of the funds being spend in the current transaction, more specifically, the transactions which output the funds being now spent. Moreover a vin describing sourced funds also includes an ECDSA signature on the script of the transaction

2. **Vout**: Each Vout included in a transaction essentially describes one outputted amount which may be spent. That is, a Vout describes a way of spending some amount of the source funds.
Before we discuss shielded transactions there are two cryptographic components we need to briefly consider.

**Zero Knowledge Commitments**

Zero Knowledge commitments are the method used by Zcash to show ownership of coins being spent. In a sense, when shielded amounts are created, a random values is “committed to”. In unshielded transactions, users verify their ownership of an amount by signing the input transaction. When attempting a private transaction a signature protocol does not work since doing so would inherently be less than perfectly anonymous. Instead, a commitment scheme is used, and a user proving their ability to spend an amount comes from them revealing the opening value of the commitment.

**zk-SNARK**

zk-SNARK is a particular zero knowledge proof system which is used in Zerocash & Zcash to allow a user to prove their ownership of a coin without revealing who they are. Consider traditional bitcoin where the spender of a coin must sign a transaction to prove they had the authority to spend the coins, creating a signature necessitates including the public key of the user which of course can be used to deanonymise the user. The zk-SNARK proof system is used here in conjunction with the zero knowledge commitment scheme to allow users to prove they have the authority to spend coins without revealing their identity. We will not consider any cryptanalysis on the zero knowledge proof scheme as it is much too complex and is beyond our scope.

zk-SNARK stands for Zero Knowledge Succinct Non-interactive Argument of Knowledge.

In particular, Zcash uses the “lib-snark” C++ implementation of this zero knowledge proving scheme.
Shielded Transactions

Shielded transactions are significantly more complex than unshielded ones. Shielded transactions are often referred to as “JoinSplits”. A shielded transaction can be one of three formats either

- Transferring money from shielded inputs to shielded outputs. (In this case there is no information about the amounts revealed, the transaction could be for 10ZEC or 10000ZEC we have no idea)

- Transferring money from unshielded inputs to shielded outputs. (In this case we can see how much money is consumed by the transaction but we don’t know to which outputs it went)

- Transferring money from shielded inputs to unshielded outputs. In this case we cannot see where the funds are coming from but we can see to which address the funds are sent to.

Figure 2: Example of (shielded → shielded) tx from explorer.zcha.in

Figure 3: Example of (unshielded → shielded) tx from explorer.zcha.in

Figure 4: Example of (shielded → unshielded) tx from explorer.zcha.in
Precise format of shielded transactions

As mentioned previously, shielded transactions can be of 3 types. The json transaction shown above is a shielded transaction sending shielded inputs to unshielded outputs. Observe that array of vin objects is empty but there is 2 vout objects as well as a vjoinsplit.
VJoinSplit

VJoinSplit objects describe the movement of shielded funds. Like unshielded transactions where there may be many vins and many vouts, a shielded transaction may have many vjoinsplits. A JoinSplit will always have exactly 2 coin inputs and 2 coin outputs.

The fields of a JoinSplit are as follows:

- **Nullifiers**: Each JoinSplit will have 2 nullifier values which correspond to the opening values to two previous coin commitments.

- **Commitments**: Each JoinSplit will have 2 commitments which are the coin commitments for the 2 outputs coins of that JoinSplit.

- **Proof**: Each JoinSplit contains one proof, which is the representation of the zk-SNARK zero knowledge proof which proves to any transaction verifier that the transaction is valid without spoiling the anonymity of the transaction sender or receiver.

- **One time Public Key**: this is a one time ephemeral curve Curve25519 public key used to perform in band symmetric key agreement with the receiver of funds.

- **Random Seed**: this is the random seed value for the new coins, simply seeding the coin commitments randomly for correct security.

- **Ciphertexts**: These are the 2 encryptions of the outputs of this transaction. i.e. the encryptions of the revealing values of the outputted coins. They are encrypted under a shared key computed using the above one time public key

- **Macs**: Message authentication codes for the above encryptions

These values facilitate the sending of funds from one address to another without revealing those addresses.
The Zcash Node software

The official Zcash node software is a large and cumbersome program which is difficult to install. Due to it’s age there is not a lot of documentation available yet. It time consuming to both install the program and to interpret which commands were needed for our task.

Only minimal install instructions are provided [1] and only for Debian Linux.

The zcash node software suite consists of two programs which we utilise:

- zcashd
- zcash-cli

zcashd is essentially the zcash node software and needs to run for some time in order to retrieve and verify all the blocks and transactions on the blockchain.

One configuration change is needed to the zcashd config file:

The config file is usually located at /home/<user>/.zcash/zcash.conf and requires the following line to be added:

```text
txindex=1
```

This sets the node to fully index all of the transactions so that they can be queried properly.

After successfully installing, we came upon the two commands we needed to obtain the appropriate data.

```text
zcash-cli getrawtransaction <transaction_id>
```

This command returns the raw transaction data from the provided transaction id

```text
zcash-cli decoderawtransaction <raw_transaction>
```

This command returns a human readable version of the provided raw transaction

Mining Zcash data

Our strategy for mining the data was to run the full zcash node and query the node for transaction data. We wrote a python program which queries the node through zcash-cli and organises the data in a well structured database.

While we mined all the data from the blockchain, including the data from shielded transactions, we only considered unshielded transactions in our analysis.

The repository for my data mining software is here: https://github.com/KillianDavitt/ZcashDataMiner
Signatures and checking for randoms

The zcash unshielded ECDSA signatures were decoded with significant guidance from [5] the similarity between bitcoin and unshielded zcash transactions means that the information provided is quite applicable.

The ECDSA signature (R,S) is contained in the “asm” attribute, inside the “vin_script” table.

The signature is DER encoded. It has a special structure, as follows:

```
CREATE TABLE vin_script (
    script_id INTEGER NOT NULL,
    vin_id INTEGER NOT NULL,
    asm VARCHAR,
    hex_script VARCHAR,
    req_sigs INTEGER,
    type_script VARCHAR,
    transaction_id INTEGER NOT NULL,
    PRIMARY KEY (script_id),
    FOREIGN KEY(vin_id) REFERENCES vin (vin_id),
    FOREIGN KEY(transaction_id) REFERENCES "transaction" (tx_id)
);
```

The ECDSA signature (R,S) is contained in the “asm” attribute, inside the “vin_script” table.

The signature is DER encoded. It has a special structure, as follows:

```
304502210d52330113cc033ccbb1aaa3b759e9696c216e802922e5f1902cd5ada69c612e50220
5780205319dcebb05ebe3e34323a523e32e82653f09f81253ddcc08a801e9d42d01
```

‘30’ indicates that a compound structure starts there. As a signature is composed of 2 parts, R and S, it is considered as a compound. ‘45’ is the total length of the compound structure. After that, the contents of the compound starts.

Then, it is the same procedure for S. ‘02’ indicates that S is an integer.

‘20’ is the byte-length in hexadecimal of S, i.e. 32 bytes in decimal.

‘5780205319dcebb05ebe3e34323a523e32e82653f09f81253ddcc08a801e9d42d01’ is S.

‘01’ is not part of the signature. It is added by Zcash and indicates what fields of the transaction are signed. 01 means “all”.  

By parsing “asm” for every transaction, we built an SQL table containing the fetched values of R and the associated transaction id. We can see in Figure 1 that transaction_id is the 10th attribute of the vin_script table.

To output the duplicate randoms, we used the SQL query:

```
SELECT r, COUNT(*) c FROM signature GROUP BY r HAVING c > 1;
```

Even with 20 million signatures as input, no duplicate randoms were printed out.
**Why didn’t it work & Some stats**

Zcash was initially launched on 28 October 2016 which could be considered to be quite recent. Obviously the newer a cryptocurrency is the less transactions there are. Zcash also has significantly less users than larger currencies like Bitcoin and Ethereum. Both of these facts mean that there is much less opportunity to find repeated randoms or other cryptographic anomalies

<table>
<thead>
<tr>
<th></th>
<th>Shielded</th>
<th>Unshielded</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Transactions</td>
<td>309,258</td>
<td>2,500,00</td>
</tr>
<tr>
<td>Number of spends</td>
<td>346,114</td>
<td>39,834,064</td>
</tr>
</tbody>
</table>

13% of transactions are shielded

0.861% of spends are shielded

**Reuse of Public Keys**

Upon suggestion of Dr. Courtois, we changed our focus to attempt to collected reused public keys. In bitcoin, it is highly discouraged from using the same public key (address) to receive multiple transactions. Doing so has significant risks to anonymity and security.

Since Zcash’s unshielded transactions behave similarly to bitcoin transactions we attempted to search for reused public keys in the zcash blockchain. This process did not require significant extra effort as all the data needed to determine reused public keys was already in our database.

Above, in our discussion of signatures in vin scripts. We determine how the signature values are contained in the first half of the asm of the vin script value. The second half of the vin script asm, contains the public key as can be seen below

![Figure 5: The format of zcash asm data](https://github.com/KillianDavitt/ZcashDataMiner/blob/master/pubkey_from_asm.py)

A simple python script was required to extract the public key information from our transaction database.

https://github.com/KillianDavitt/ZcashDataMiner/blob/master/pubkey_from_asm.py

After the public keys and transaction id pairs were collected in a file. Another simple script was required to collate them, remove duplicate transaction id’s and count the actual repetitions of user public keys. The following script did this.

https://github.com/KillianDavitt/ZcashDataMiner/blob/master/count_dupes.py

**Results of repeated public keys**

The collection of repeated user public keys proved to be much more successful than the collection of repeated signature random numbers. In total, approximately 20 million signatures used public keys which had already been used. It is immediately clear that reuse of public keys is completely widespread in the Zcash blockchain.
The most reused public key:

035556ffd87eda7df56cf4a1f180022e22572920d70994cb9139ba973f2a105d71

This public key was used to sign 41,138 transactions. This is an enormous number and this account obviously contains a large concentration of Zcash activity. We derived the account address to be:

t1VpYecBW4UudbGcy4ufh61eWxQCoFaUrPs and it can be viewed here:

https://explorer.zcha.in/accounts/t1VpYecBW4UudbGcy4ufh61eWxQCoFaUrPs

This particular blockchain explorer has identified this as a mining pool account “flypool” and this explains why there are so many transactions going through this account.

It is interesting to note that there appears to be no advice relating to the reuse of public keys provided by the Zcash team. And indeed, the official Zcash client is also ambiguous and does not encourage the use of multiple public keys. It is possible that this is a large factor in the reason why so many public keys are reused.

Other potential Cryptographic events

The main focus of this project was to attempt to find repeated random values in ECDSA signatures, however there are a few more potential areas of cryptographic interest particularly in the JoinSplit objects. As is discussed earlier, each JoinSplit includes a one time public key as well as a random seed and 2 ciphertexts. Of course, since the random seeds are public they could be examined for collisions. The one time public keys could also be examined for collisions.

As of today, we don’t believe that there exists enough shielded transactions on the blockchain to make these examinations worthwhile. In fact we checked for collisions of the “random seed” values and of the 346,114 checked, there were no duplicates.

One would imagine that if our software was run again in the future after the Zcash blockchain grows, there would be a much higher chance of finding both the unshielded repeated randoms and possibly more interesting repitions in the VJoinSplit object.

Conclusion

In conclusion, our search for duplicated random values in signatures was not successful. However, many many duplicates public keys were found. While it appears the designers of Zcash were careful to avoid any problems with repeating randoms, they did not properly discourage reuse of public keys.

We also suppose that one possible reason that no random collisions were found is the uniformity of clients. Currently most users of Zcash are encouraged to use the official client and there does not exist a huge number of alternative clients as there does with other coins like bitcoin. However as Zcash grows there may be potential for more errors in randoms to occur. In addition, as Zcash grows, it should be beholden to the Zcash team to discourage more strongly, the reuse of public keys.
References


